TDD TRAINING – DAY 1

SHEKHAR GULATI

AGENDA – DAY 1

1. Introduction to TDD
2. Evolutionary Design and Coding
3. Using Mock Objects
4. Unit Testing Best Practices

TRAINING WILL COVER VARIOUS ASPECTS

1. Refactoring
2. SOLID principles
3. Design patterns
4. Version control
5. Working effectively with IDE – Eclipse
6. Being pragmatic

WHAT WE WILL NOT COVER?

1. Web framework specific testing
2. Database testing
3. Web service testing
4. Functional testing frameworks like Selenium

THERE ARE NO STUPID QUESTIONS ONLY STUPID ANSWERS SO PLEASE FEEL FREE TO ASK QUESTIONS

WHY SHOULD WE TEST SOFTWARE?

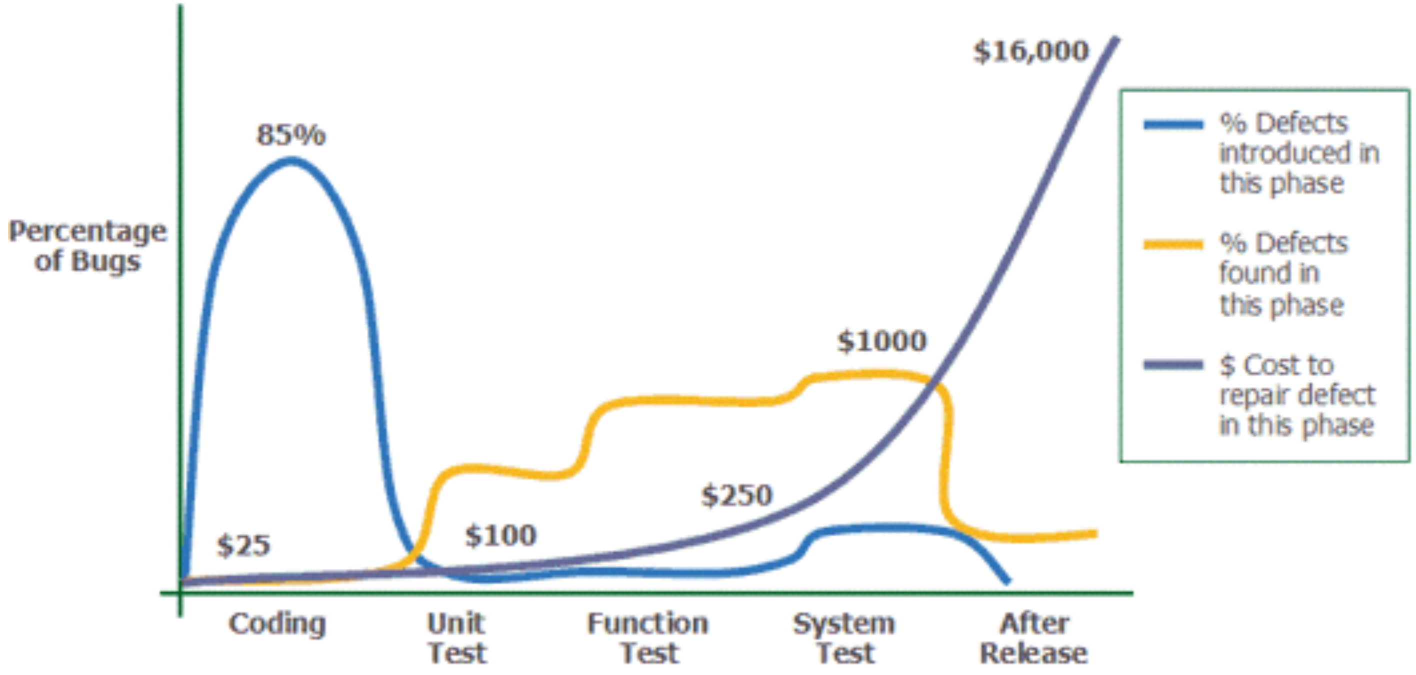
SOME OF THE MOTIVATIONAL FACTORS:

1. To verify that our program produce correct output
2. To check if we meet end user expectations
3. To clarify our intentions on what the program should do
4. To reduce bugs ;)
5. To be sure that we can ship software

TESTING COULD HELP:

1. Fix bugs a bit faster
2. Fix bugs a bit cheaper
3. Make software a bit better – measurable, testable, indicative

FASTER AND CHEAPER

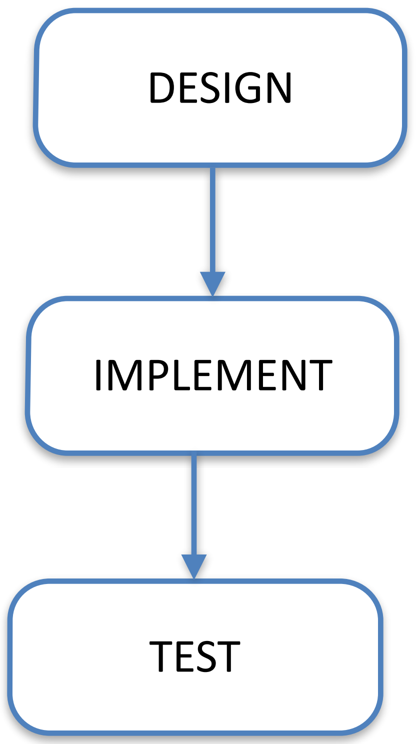


http://blogs.windriver.com/graham/2010/01/service-and-repair-is-not-the-only-option.html

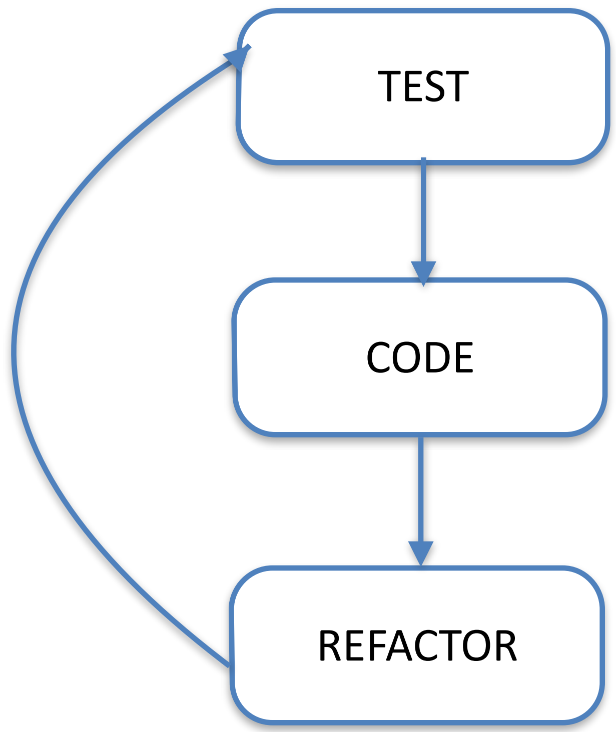
THERE ARE TWO SOLUTIONS

1. Test After Development
    1. This includes writing tests after you have written code, functional automation tests written after coding, etc
    2. Hiring a fresher and asking her to write tests
2. Test First Development
    1. Tests are written before there is any working code for that functionality
    2. Developer who writes functionality writes tests

TEST AFTER VS TEST FIRST



TEST AFTER



TEST FIRST

TDD as a DESIGN TOOL

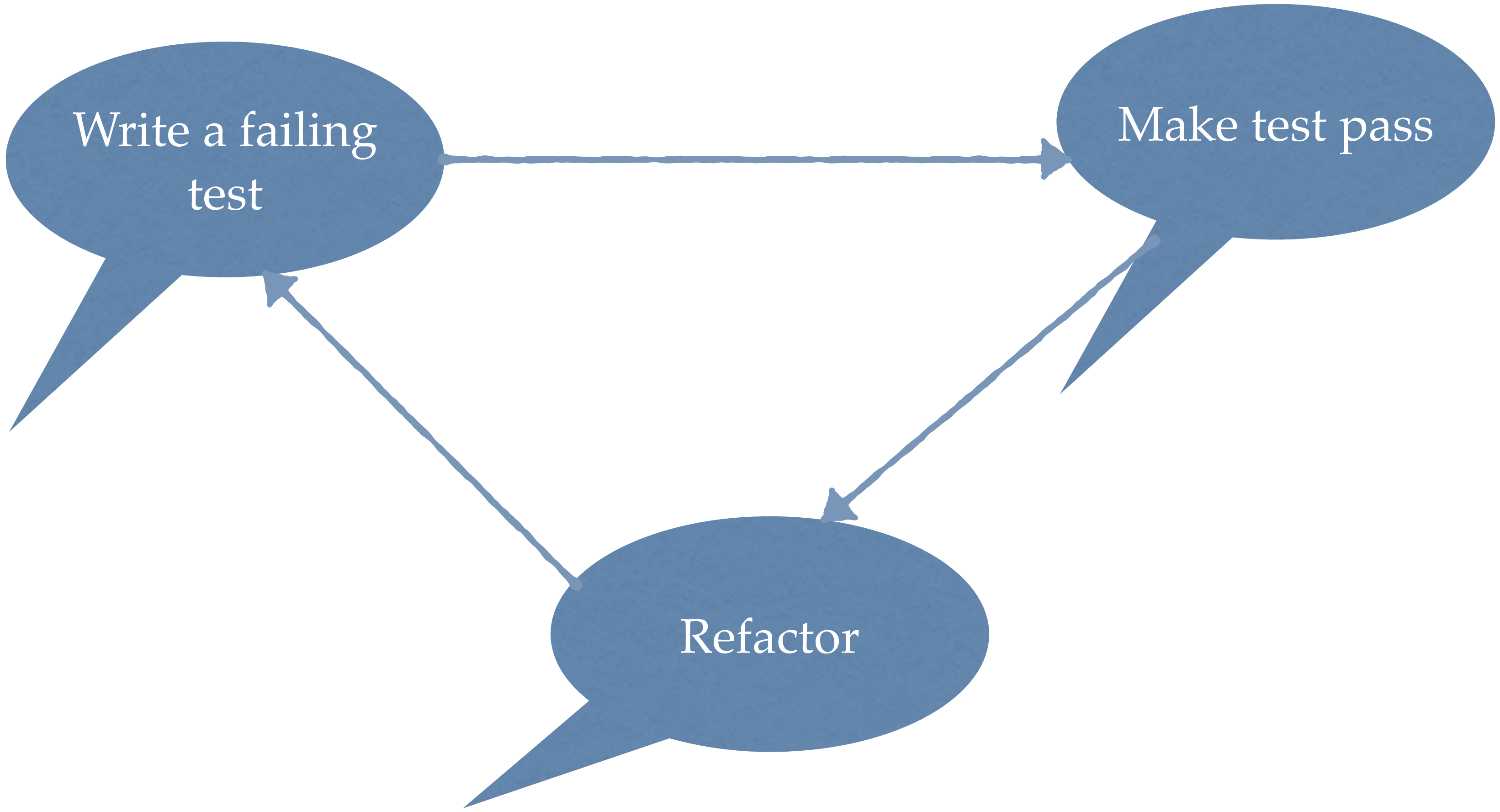
GOALS OF TEST AUTOMATION

1. Improve software quality
2. Help us understand the System Under Test(SUT)
3. Improve the overall design of SUT
4. Faster feedback cycle —> Feedback driven development
5. Minimal maintenance with time

TEST DRIVEN DEVELOPMENT

1. Write **one test** that describes a new behaviour
2. Write **minimalistic code** to make it pass
3. Improve design by **refactoring code** and **tests**

TDD IN NUTSHELL



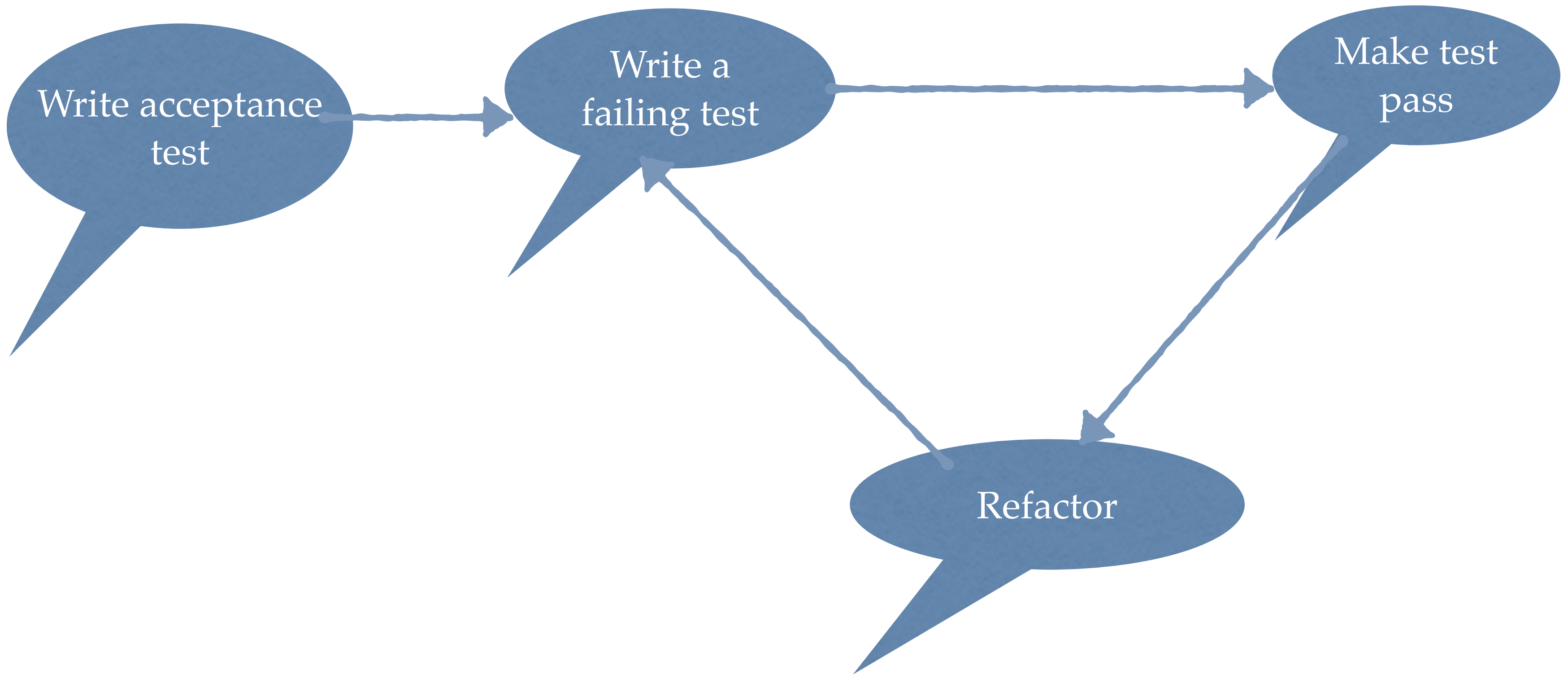
GOLDEN RULE OF TDD

Never write a single line of production code without a failing test

HOW TO APPROACH A USER STORY

1. First write a failing acceptance criteria that exercise the functionality code
2. Then repeat failing unit test -> code to pass test -> refactor cycle

USER STORY VIEW



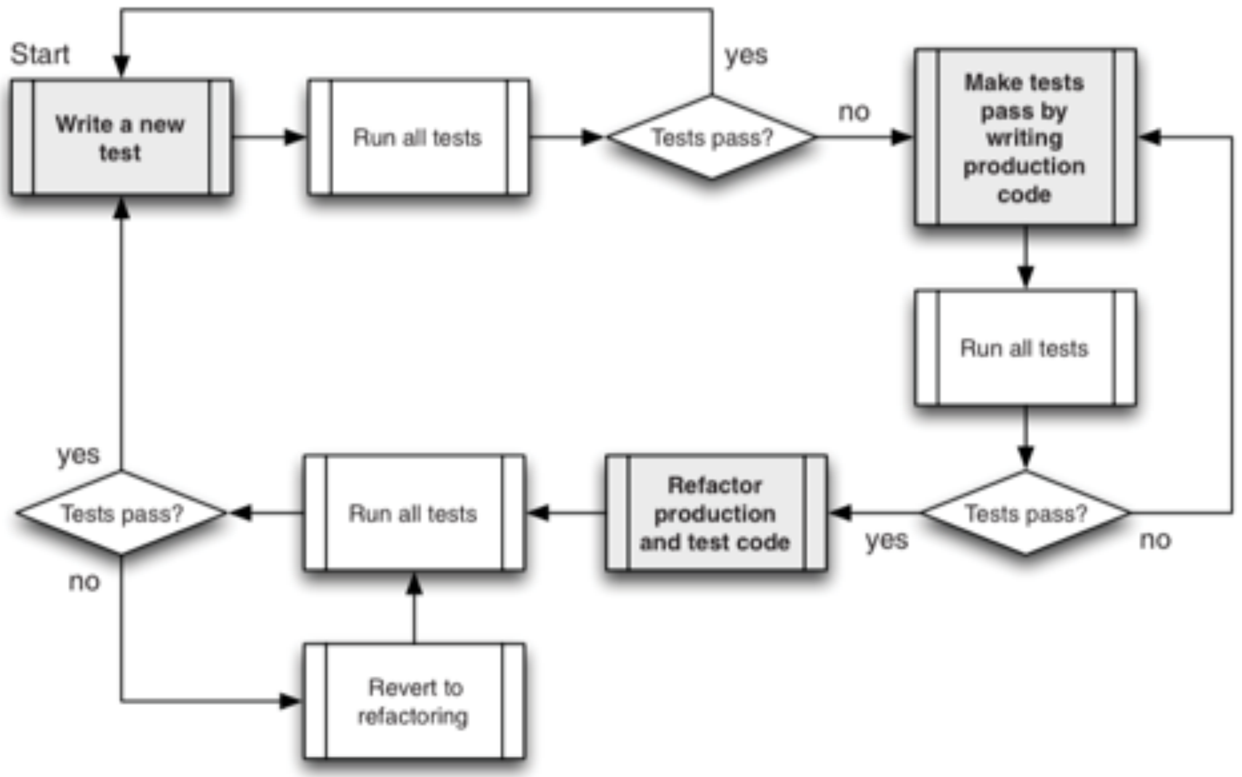
TDD WITH BDD GLASSES

1. Start with a failing acceptance test that describes the behaviour of the system from the customer's point of view
2. They are then used to communicate with business stakeholders

TDD IN ACTION – DEMO

1. Setup machines
2. Open Eclipse
3. Import Maven project
4. Understanding the problem domain
5. Looking at code in the master
6. Writing first user story using TDD

ESSENCE OF TDD



WHY TEST FIRST?

WHY TEST FIRST DEVELOPMENT?

1. Focus on what rather than how
2. Meet end user expectation by thinking in terms of the consumer of the code
3. Gives confidence to change implementation without affective end user
4. Leads to minimalistic design
5. Enables safe refactoring of virtually all your code

TDD CAN HELP YOU CHOSE RIGHT DATASTORE FOR YOUR PROBLEM

1. If you follow test first software development approach you start to defer decisions to the last responsible moment. This can help you decide at a later stage which database would suit your needs as you use mock or in-memory data structure it will become clear what kind of access patterns are there. If all you are doing is accessing based on a key. It would be a better fit for key value store like Redis rather than RDBMS

TDD GIVES FEEDBACK ON BOTH IMPLEMENTATION AND DESIGN

Write tests

1. Acceptance criteria(design)
2. Loosely coupled components(design)
3. Executable description of what code should do(design)
4. Adds to a complete regression suite(implementation)

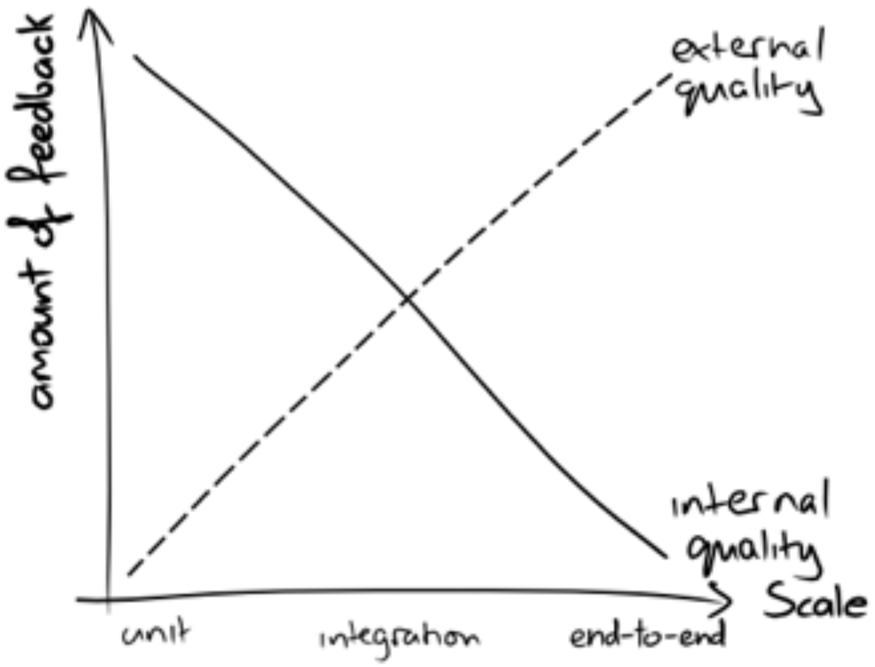
Run test

1. Detect errors early in the lifecycle(implementation)
2. let us know when we've done enough(design)

LEVELS OF TESTING

1. Acceptance testing – does the whole system work
2. Integration testing – does our code work against code we can't change
3. Unit testing – does function or object work as expected

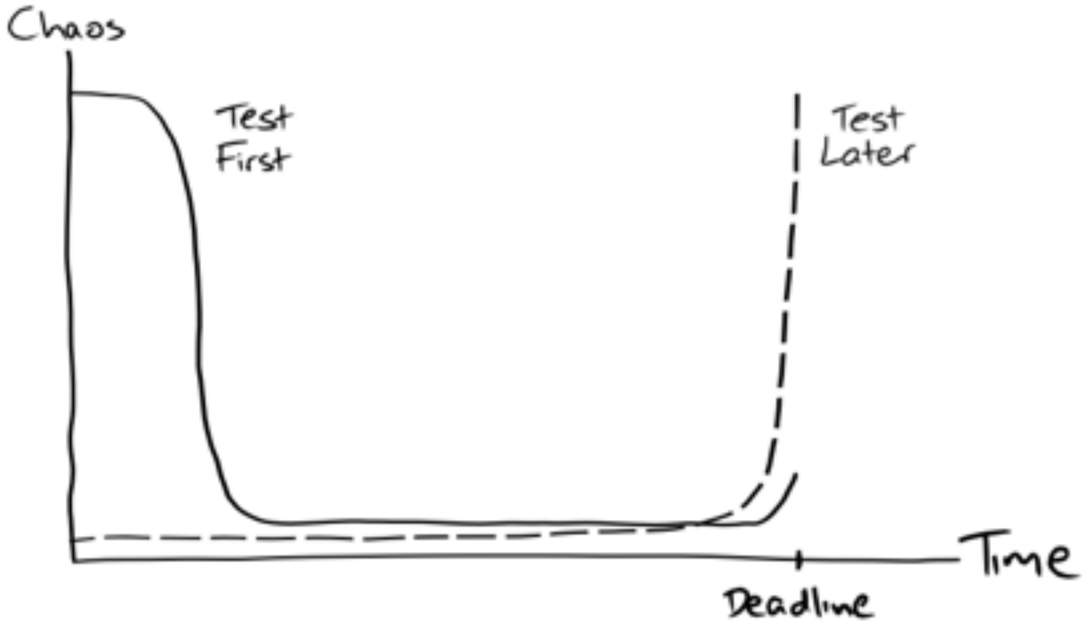
EXTERNAL VS INTERNAL QUALITY



IMPORTANCE OF EARLY END-TO-END TESTING

1. Early end-to-end testing is the foundation for continuous deployment
2. Things that pain should be done more often
3. Expose uncertainty early in the development cycle. Fail fast is the mantra
4. Integration becomes much easier and troublesome if you make end-to-end testing a priority

TEST FIRST EXPOSE UNCERTAINTY EARLY



JUNIT 101

JUNIT

1. JUnit uses reflection to invoke the methods of a test class.
2. All the methods should be annotated using @Test annotation
3. A test method should be
    1. public
    2. not return anything
    3. not take any parameters
4. Each time JUnit runs a test method it creates a new instance of the Test class. This means each test method invocation is independent of others
5. Test should be isolated and not depend on the state

JUNIT @RUNWITH ANNOTATION

1. The @RunWith annotation specifies how JUnit find tests in the test class.
2. JUnit comes with few test runners
   1. Parameterised test runner
   2. Default test runner

JUNIT SUPPORTS TWO TYPES OF ASSERTION STYLE

1. Classic style like assertTrue, assertEquals
2. Expressive assertions using Hamcrest and assertThat

JUNIT DIFFERENCE BETWEEN ERROR AND FAILURE

1. Failure when expectation breaks
2. Error when code does not compile or any exception thrown by code

USING MULTIPLE BEFORE METHOD

1. Use @Before methods with clear names like createShoppingCart(), createBooks().
2. Execution order should not matter for @Before methods

WRITING DECLARATIVE TESTS

ASSERT THAT AND HAMCREST

Demo – Convert a test to Hamcrest based test

PROPERTIES OF A GOOD TEST

1. Fast
2. Isolated
3. Repeatable
4. Self-validating
5. Timely

NAMING TESTS
–
USING CLEAR TEST NAMES

NAMING IS A HARD PROBLEM

1. Using proper class names
2. Using behavioural names for tests
3. Using domain specific names for variables, classes, methods

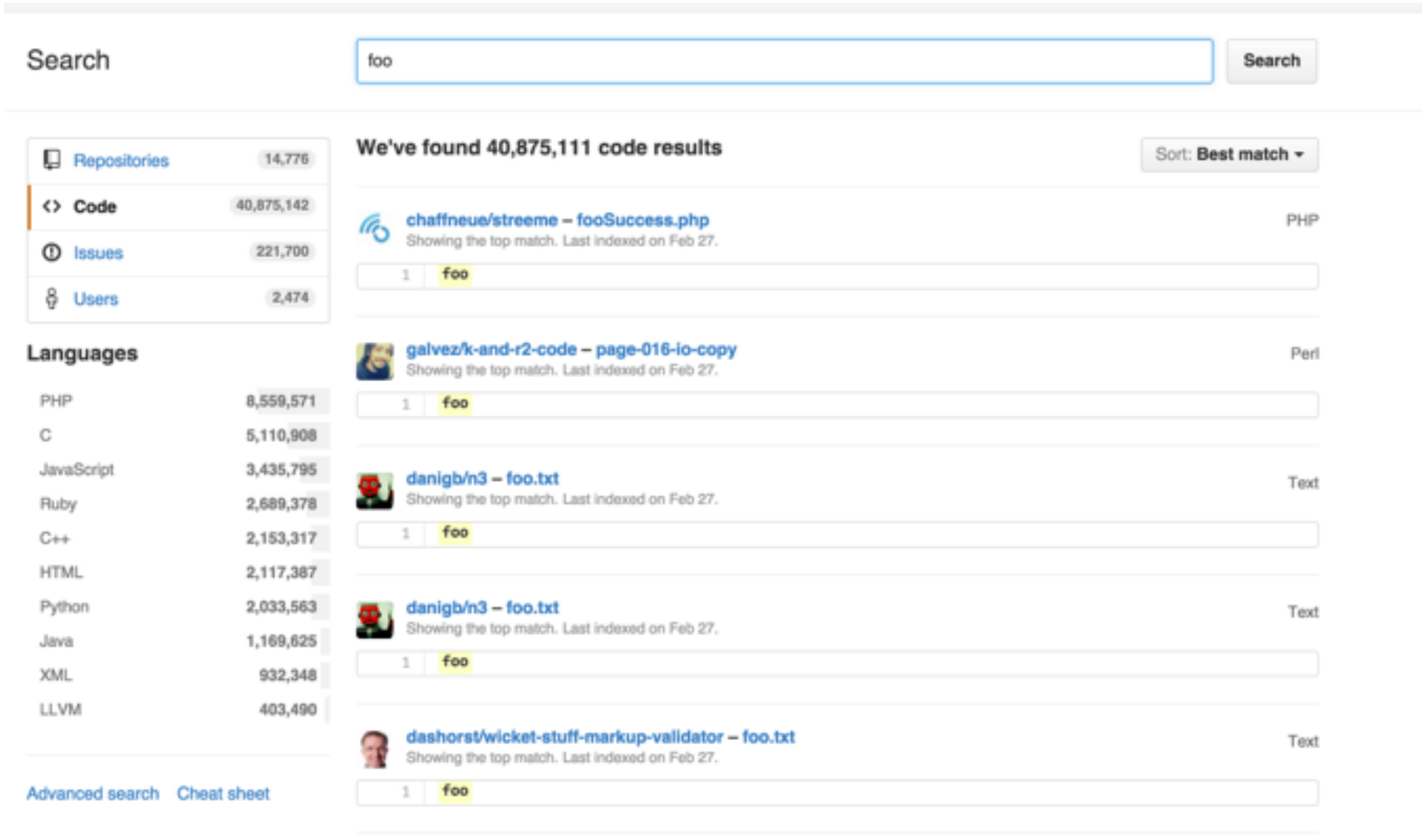
http://www.slideshare.net/pirhilton/how-to-name-things-the-hardest-problem-in-programming

NAMING TESTS

givenSomeContext_WhenDoingSomeWork_ThenSomeResultOccurs

whenDoingSomeWork_ThenSomeResultOccurs

USAGE OF FOO



Search

foo

Search

Repositories	14,776
Code	40,875,142
Issues	221,700
Users	2,474

Languages

PHP	8,559,571
C	5,110,908
JavaScript	3,435,795
Ruby	2,689,378
C++	2,153,317
HTML	2,117,387
Python	2,033,563
Java	1,169,625
XML	932,348
LLVM	403,490

Advanced search Cheat sheet

We've found 40,875,111 code results

Sort: **Best match**

chaffneue/streeme – fooSuccess.php — PHP
Showing the top match. Last indexed on Feb 27.
```
foo
```

galvez/k-and-r2-code – page-016-io-copy — Perl
Showing the top match. Last indexed on Feb 27.
```
foo
```

danigb/n3 – foo.txt — Text
Showing the top match. Last indexed on Feb 27.
```
foo
```

danigb/n3 – foo.txt — Text
Showing the top match. Last indexed on Feb 27.
```
foo
```

dashorst/wicket-stuff-markup-validator – foo.txt — Text
Showing the top match. Last indexed on Feb 27.
```
foo
```

What is the worst ever variable name?

```
data
```

What is the second-worst name?

```
data2
```

What is the third-worst name ever?

```
data_2
```

BAD NAMES FOR CLASSES

1. *Manager
2. *Service
3. *ServiceImpl

NAMING TESTS

1. Test names should correspond to behaviour. Never write tests for methods always test behaviours. You might need to combine methods to form a test for behaviour. This was clear when we were writing tests for shopping cart

NAMING STRATEGY

1. There are different test naming strategies like doingSomeOperationProducesSomeResult or someResultUnderSomeContext or you can follow BDD nomenclature I.e. GivenWhenThen or whenThen

RISE OF BEHAVIOUR DRIVEN DEVELOPMENT

BDD IS TDD DONE RIGHT

1. Write tests that tests the behaviour of code in question rather than testing the method
2. Each test method talks about one behaviour and normally one line

TESTS SHOULD BE DECLARATIVE AND TEST BEHAVIOUR NOT METHOD

Test behaviours not methods. This mean you might have to test multiple methods to define interesting behaviour for example instead of testing balance() method in an ATM class you should mix withdrawal functionality together and then you would see interesting behaviour test cases

BAD TESTS

1. when tests fail when you refactor the code i.e. change the internals of your code. This means you tests are not behavioural but depends on the implementation.
2. Lets suppose we change the ShoppingCart to use List instead of Map and our test used Map interface this would break the tests. Hence bad test

ORGANIZE YOUR TEST CASE

ORGANISE TESTCASE USING AAA

1. Arrange
2. Act
3. Assert

WRITING EFFECTIVE TESTS

WRITING TESTS

1. Tests should never depend on implementation details they should always test behaviour and take outside in approach. Work on public api
2. Every time you feel the urge to write tests for complex private method that is the case where u should extract the complex behaviour to another class
3. Always make sure you can run entire test suite in seconds to get quick feedback on your implementation if tests fails then fix tests first before doing anything further. Always keep green bar

WHEN WRITING UNIT TESTS THINK ABOUT CARDINALITY

1. Test for
   1. 0 – checkout when no items in cart
   2. 1 – checkout when only one item in cart
   3. n – checkout with items more than one

BUILDING TDD HABIT

1. Fail build when test fails
2. Enforce code coverage
3. Reward people
4. Shame for build failures during retro
5. Always run unit tests before committing
6. Use something like https://infinitest.github.io/ to continuously run your unit tests
7. Enforce tests as part of definition of done

WHAT TO TEST – RIGHT BICEP

1. Right – Are the results right?
2. Boundary – Test boundary condition
3. Inverse – Check inverse relationships
4. Forcing error conditions
5. Performance characteristics

REFACTORING TESTS

REFACTORING TESTS – TEST SMELLS – CATCHING EXCEPTIONS

1. Don't catch exceptions in test methods unless you want to assert that exception. There are better ways to work with assertions so you can always avoid catching exceptions
2. JUnit itself catches the exceptions and mark test as error.

TEST SMELL – CHECKING FOR NULL

1. At times you can get rid of not null checks in tests by just asserting the value. If the value is null your test will fail

TEST SMELL – MISSING ABSTRACTIONS

1. Use of custom domain specific matchers like IsShoppingCartSizeWithSizeMatcher
2. abstract out smaller meaningful methods to avoid reader from reading all the code and making sense out of it

TEST SMELL – MULTIPLE ASSERTIONS

1. If you are using multiple asserts in your test case that is a sign for one test testing multiple behavior. This happended when we were asserting items() and size() in the canAddMultipleItemsInOneGo test method. If you have to write multiple asserts for the return value, it might be better to split that into its own assert method

TEST SMELL – TEST MISSING CLEAR AAA SEPERATION

1. Looking at the test it should be clear AAA abstraction

TEST SMELL – TEST DATA DOES NOT CLEARLY TELL THE INTENT

1. Using coupon code like “valid_coupon_code” or “expired_coupon_code” can clearly tell a reader the intent of the test. Reduce the mental burden on the developer who will later maintain and use the test cases.

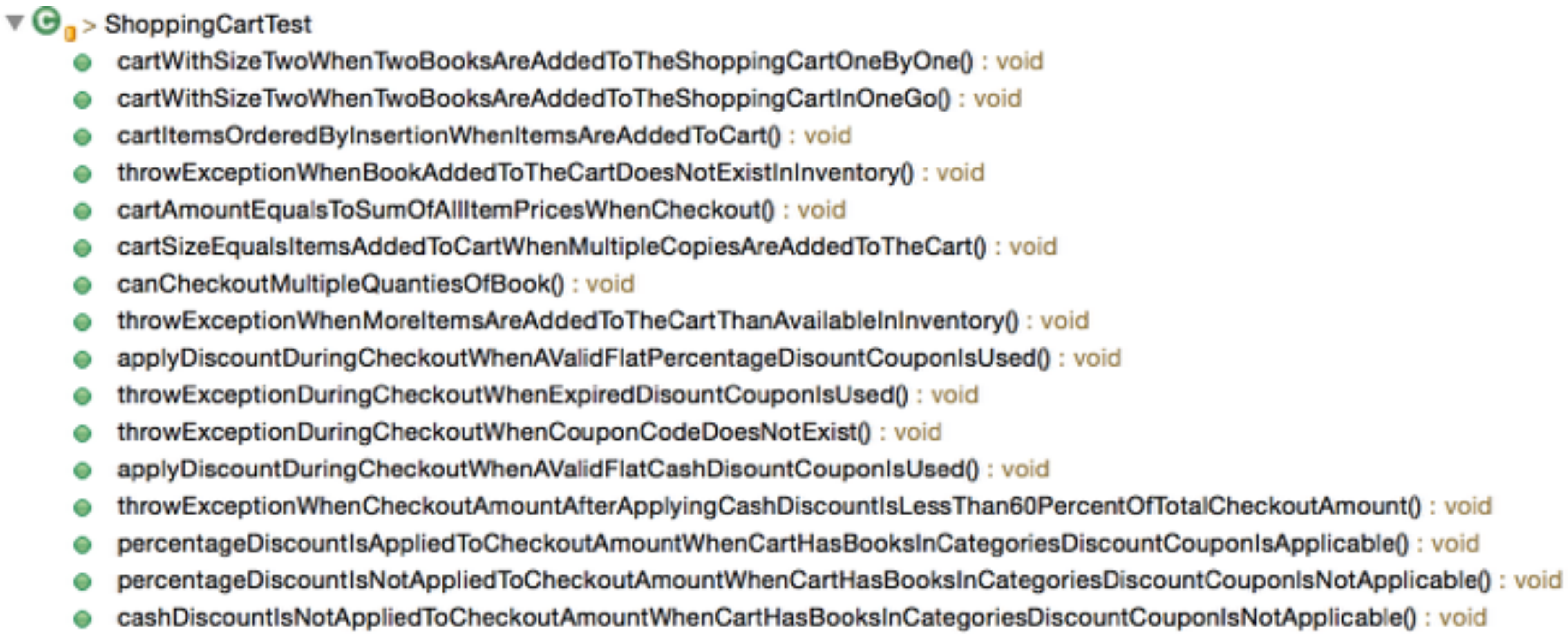
VERSION CONTROL AND TESTS

VERSION CONTROL AND TESTS

As soon as your test pass, commit it to the version control system. You can use features like squash merge to create a single commit later on if required. You can always keep all commits in your local branch

EXECUTABLE DOCUMENTATION

TDD SIDE EFFECT – EXECUTABLE SPECIFICATION



```
▼ ShoppingCartTest
    cartWithSizeTwoWhenTwoBooksAreAddedToTheShoppingCartOneByOne() : void
    cartWithSizeTwoWhenTwoBooksAreAddedToTheShoppingCartInOneGo() : void
    cartItemsOrderedByInsertionWhenItemsAreAddedToCart() : void
    throwExceptionWhenBookAddedToTheCartDoesNotExistInInventory() : void
    cartAmountEqualsToSumOfAllItemPricesWhenCheckout() : void
    cartSizeEqualsItemsAddedToCartWhenMultipleCopiesAreAddedToTheCart() : void
    canCheckoutMultipleQuantiesOfBook() : void
    throwExceptionWhenMoreItemsAreAddedToTheCartThanAvailableInInventory() : void
    applyDiscountDuringCheckoutWhenAValidFlatPercentageDisountCouponIsUsed() : void
    throwExceptionDuringCheckoutWhenExpiredDisountCouponIsUsed() : void
    throwExceptionDuringCheckoutWhenCouponCodeDoesNotExist() : void
    applyDiscountDuringCheckoutWhenAValidFlatCashDisountCouponIsUsed() : void
    throwExceptionWhenCheckoutAmountAfterApplyingCashDiscountIsLessThan60PercentOfTotalCheckoutAmount() : void
    percentageDiscountIsAppliedToCheckoutAmountWhenCartHasBooksInCategoriesDiscountCouponIsApplicable() : void
    percentageDiscountIsNotAppliedToCheckoutAmountWhenCartHasBooksInCategoriesDiscountCouponIsNotApplicable() : void
    cashDiscountIsNotAppliedToCheckoutAmountWhenCartHasBooksInCategoriesDiscountCouponIsNotApplicable() : void
```

USING MOCK OBJECTS

STUB OBJECT

1. A stub is an empty container, that represent an Object

WHAT IS MOCKING?

"A mock object is an object that takes the place of a 'real' object in such a way that makes testing easier and more meaningful, or in some case possible at all"

by Scot Bain – Emergent Design

MOCKS

MOCKS

WHY USE MOCKING?

1. Isolate your SUT
2. To build against interfaces and contracts
3. Building against missing integration pieces
4. To control data and expectations
5. Mock components whose behaviour is undesirable or hard to control

WHY MOCK OVER STUB?

1. To specify in the test which parameters mock expected – when().thenReturn()
2. Trapping and storing the parameter passed to the collaborator method
3. Supporting the ability to verify upon completion that the stored parameters to the collaborator method contains the expected parameters – verify() method

MOCKITO 101

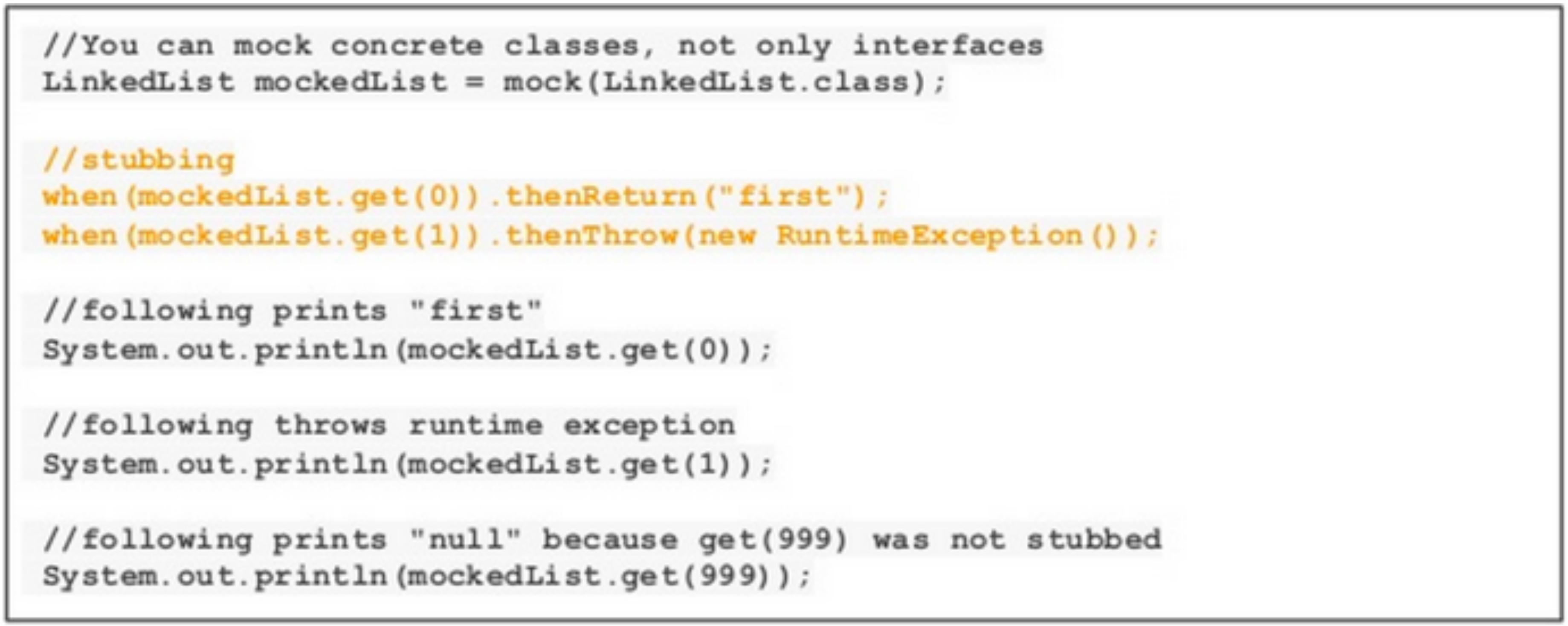
MOCKITO

1. Mockito provides a framework for interaction testing
2. It is a Java framework allowing the creation of test double objects in automated unit tests
3. Interaction testing verifies the interaction between objects
    1. Did my ShoppingCart call my inventory

TEST DOUBLE

1. Test Double is a generic term for any case where you replace production object for testing purposes

EXAMPLE



```java
//You can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.class);

//stubbing
when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(1)).thenThrow(new RuntimeException());

//following prints "first"
System.out.println(mockedList.get(0));

//following throws runtime exception
System.out.println(mockedList.get(1));

//following prints "null" because get(999) was not stubbed
System.out.println(mockedList.get(999));
```

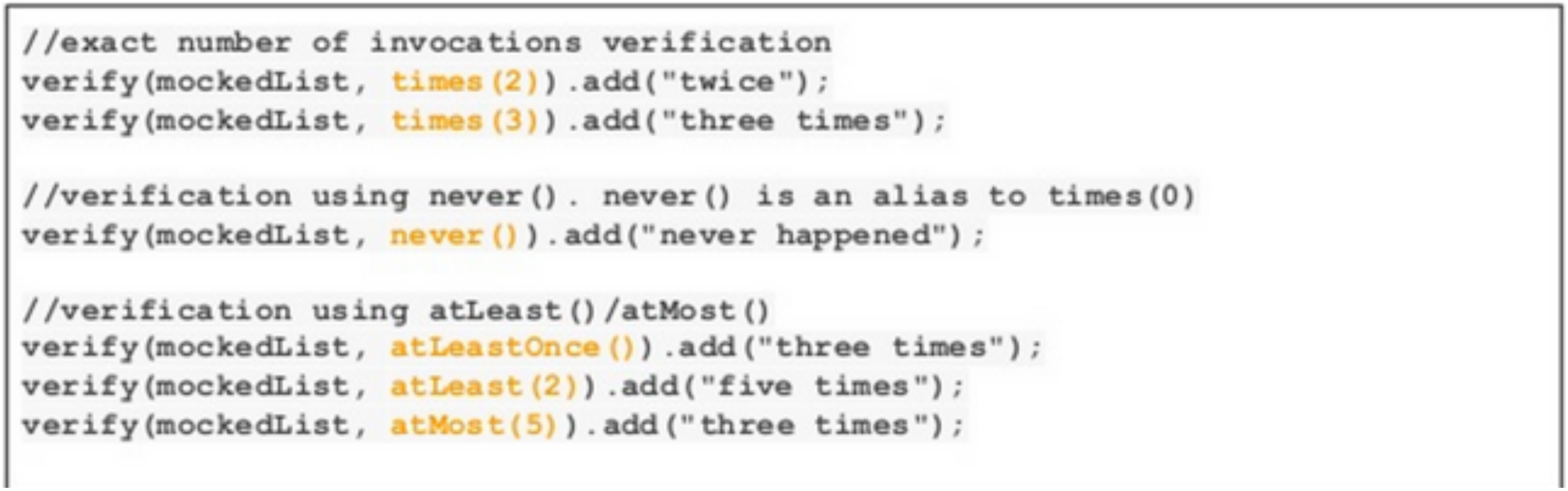
ARGUMENT MATCHER

Note that if you are using argument matchers, **all arguments** have to be provided by matchers

```
verify(mock).someMethod(anyInt(), anyString(), eq("third argument"));
//above is correct - eq() is also an argument matcher


verify(mock).someMethod(anyInt(), anyString(), "third argument");
//above is incorrect - exception will be thrown because third argument is
//given without an argument matcher
```

VERIFYING EXACT NUMBER OF INVOCATIONS / AT LEAST X / NEVER



```
//exact number of invocations verification
verify(mockedList, times(2)).add("twice");
verify(mockedList, times(3)).add("three times");

//verification using never(). never() is an alias to times(0)
verify(mockedList, never()).add("never happened");

//verification using atLeast()/atMost()
verify(mockedList, atLeastOnce()).add("three times");
verify(mockedList, atLeast(2)).add("five times");
verify(mockedList, atMost(5)).add("three times");
```